Graph cheatsheet for coding interviews

Introduction

A graph is a structure containing a set of objects (nodes or vertices) where there can be edges between these nodes/vertices. Edges can be directed or undirected and can optionally have values (a weighted graph). Trees are undirected graphs in which any two vertices are connected by exactly one edge and there can be no cycles in the graph.
Graphs are commonly used to model relationship between unordered entities, such as

- Friendship between people - Each node is a person and edges between nodes represent that these two people are friends.
- Distances between locations - Each node is a location and the edge between nodes represent that these locations are connected. The value of the edge represents the distance.

Be familiar with the various graph representations, graph search algorithms and their time and space complexities.

Learning resources

- Readings
  - From Theory To Practice: Representing Graphs, basecs
  - Deep Dive Through A Graph: DFS Traversal, basecs
  - Going Broad In A Graph: BFS Traversal, basecs
- Additional (only if you have time)
  - Finding The Shortest Path, With A Little Help From Dijkstra, basecs
  - Spinning Around In Cycles With Directed Acyclic Graphs, basecs

Graph representations

You can be given a list of edges and you have to build your own graph from the edges so that you can perform a traversal on them. The common graph representations are:

- Adjacency matrix
- Adjacency list
- Hash table of hash tables

Using a hash table of hash tables would be the simplest approach during algorithm interviews. It will be rare that you have to use an adjacency matrix or list for graph questions during interviews.
In algorithm interviews, graphs are commonly given in the input as 2D matrices where cells are the nodes and each cell can traverse to its adjacent cells (up/down/left/right). Hence it is important that you be familiar with traversing a 2D matrix. When traversing the matrix, always ensure that your current position is within the boundary of the matrix and has not been visited before.

Time complexity

|V| is the number of vertices while |E| is the number of edges.

Algorithm	Big-O
Depth-first search	O(|V| + |E|)
Breadth-first search	O(|V| + |E|)
Topological sort	O(|V| + |E|)

Things to look out for during interviews

- A tree-like diagram could very well be a graph that allows for cycles and a naive recursive solution would not work. In that case you will have to handle cycles and keep a set of visited nodes when traversing.
- Ensure you are correctly keeping track of visited nodes and not visiting each node more than once. Otherwise your code could end up in an infinite loop.

Corner cases

- Empty graph
- Graph with one or two nodes
- Disconnected graphs
- Graph with cycles

Graph search algorithms

- Common - Breadth-first Search, Depth-first Search
- Uncommon - Topological Sort, Dijkstra's algorithm
- Almost never - Bellman-Ford algorithm, Floyd-Warshall algorithm, Prim's algorithm, Kruskal's algorithm. Your interviewer likely doesn't know them either.

Depth-first search

Depth-first search is a graph traversal algorithm which explores as far as possible along each branch before backtracking. A stack is usually used to keep track of the nodes that are on the current search path. This can be done either by an implicit recursion stack, or an actual stack data structure.
A simple template for doing depth-first searches on a matrix goes like this:

```
def dfs(matrix):
 # Check for an empty matrix/graph.
 if not matrix:
  return []

 rows, cols = len(matrix), len(matrix[0])
 visited = set()
 directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

 def traverse(i, j):
  if (i, j) in visited:
   return

  visited.add((i, j))
  # Traverse neighbors.
  for direction in directions:
    next_i, next_j = i + direction[0], j + direction[1]
    if 0 <= next_i < rows and 0 <= next_j < cols:
      # Add in question-specific checks, where relevant.
      traverse(next_i, next_j)

 for i in range(rows):
  for j in range(cols):
    traverse(i, j)
```

Breadth-first search

Breadth-first search is a graph traversal algorithm which starts at a node and explores all nodes at the present depth, before moving on to the nodes at the next depth level. A queueis usually used to keep track of the nodes that were encountered but not yet explored.
A similar template for doing breadth-first searches on the matrix goes like this. It is important to use double-ended queues and not arrays/Python lists as dequeuing for double-ended queues is O(1) but it's O(n) for arrays.

```
from collections import deque

def bfs(matrix):
 # Check for an empty matrix/graph.
 if not matrix:
  return []

 rows, cols = len(matrix), len(matrix[0])
 visited = set()
 directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

 def traverse(i, j):
  queue = deque([(i, j)])
  while queue:
    curr_i, curr_j = queue.popleft()
    if (curr_i, curr_j) not in visited:
      visited.add((curr_i, curr_j))
      # Traverse neighbors.
      for direction in directions:
        next_i, next_j = curr_i + direction[0], curr_j + direction[1]
        if 0 <= next_i < rows and 0 <= next_j < cols:
          # Add in question-specific checks, where relevant.
          queue.append((next_i, next_j))

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

INFO

While DFS is implemented using recursion in this sample, it could also be implemented iteratively similar to BFS. The key difference between the algorithms lies in the underlying data structure (BFS uses a queue while DFS uses a stack). The deque class in Python can function as both a stack and a queue.

For additional tips on BFS and DFS, you can refer to this LeetCode post

Topological sorting

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.
Topological sorting is most commonly used for scheduling a sequence of jobs or tasks which has dependencies on other jobs/tasks. The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started.
Another example is taking courses in university where courses have pre-requisites.
Here's an example where the edges is an array of two-value tuples and the first value depends on the second value.

```
def graph_topo_sort(num_nodes, edges):
  from collections import deque
  nodes, order, queue = {}, [], deque()
  for node_id in range(num_nodes):
    nodes[node_id] = { 'in': 0, 'out': set() }
  for node_id, pre_id in edges:
    nodes[node_id]['in'] += 1
    nodes[pre_id]['out'].add(node_id)
  for node_id in nodes.keys():
    if nodes[node_id]['in'] == 0:
      queue.append(node_id)
  while len(queue):
    node_id = queue.pop()
    for outgoing_id in nodes[node_id]['out']:
      nodes[outgoing_id]['in'] -= 1
      if nodes[outgoing_id]['in'] == 0:
        queue.append(outgoing_id)
    order.append(node_id)
  return order if len(order) == num_nodes else None

print(graph_topo_sort(4, [[0, 1], [0, 2], [2, 1], [3, 0]]))
# [1, 2, 0, 3]
```

Essential questions

These are essential questions to practice if you're studying for this topic.

- Number of Islands
- Flood Fill
- 01 Matrix

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Breadth-first search
    - Rotting Oranges
    - Minimum Knight Moves (LeetCode Premium)
- Either search

    - Clone Graph
    - Pacific Atlantic Water Flow
    - Number of Connected Components in an Undirected Graph (LeetCode Premium)
    - Graph Valid Tree (LeetCode Premium)
- Topological sorting
    - Course Schedule
    - Alien Dictionary (LeetCode Premium)